Прямое и обратное дискретное косинусное преобразование FDCT и IDCT.

Описание алгоритма.

Двумерное дискретное косинусное преобразование является ключевым для стандартов JPEG и MPEG.

Прямое дискретное косинусное преобразования вычисляется по формуле:

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le p \le M-1 \\ 0 \le q \le N-1 \end{matrix}$$

где

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}}, & p = 0 \\ \sqrt{\frac{2}{M}}, & 1 \le p \le M-1 \end{cases}$$

$$\alpha_q = \begin{cases} \frac{1}{\sqrt{N}}, & q = 0 \\ \sqrt{\frac{2}{N}}, & 1 \le \text{q} \le \text{N-1} \end{cases}$$

Обратное дискретное косинусное преобразование вычисляется по формуле:

$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le m \le M-1 \\ 0 \le n \le N-1 \end{matrix}$$

где

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}}, & p = 0 \\ \sqrt{\frac{2}{M}}, & 1 \le p \le M-1 \end{cases}$$

$$\alpha_q = \begin{cases} \frac{1}{\sqrt{N}}, & q = 0 \\ \sqrt{\frac{2}{N}}, & 1 \le q \le N-1 \end{cases}$$

Общие детали реализации.

- Поскольку в алгоритме JPEG при дискретном косинусном преобразовании в основном используется окно 8x8, то в данной работе реализовано ДКП для двумерного массива размером 8x8.
- Так как известно, что размер окна равен 8, то для сокращения вычислительных затрат заранее вычислены значения косинусов.
- Так как при работе с изображениями в основном используются целочисленные типы данных, то входные и выходные данные для всех реализаций имеют формат int32.

Реализация без MSA с использованием чисел с плавающей запятой.

```
#define NORM 0.25
#define C 0.70711
float Cosine[8][8] = {
    {1.00000, 1.00000, 1.00000, 1.00000, 1.00000, 1.00000, 1.00000, 1.00000},
    {0.98079, 0.83147, 0.55557, 0.19509, -0.19509, -0.55557, -0.83147, -0.98079},
    {0.92388, 0.38268, -0.38268, -0.92388, -0.92388, -0.38268, 0.38268, 0.92388},
    {0.83147, -0.19509, -0.98079, -0.55557, 0.55557, 0.98079, 0.19509, -0.83147},
    {0.70711, -0.70711, -0.70711, 0.70711, 0.70711, -0.70711, -0.70711, 0.70711},
    {0.55557, -0.98079, 0.19509, 0.83147, -0.83147, -0.19509, 0.98079, -0.55557},
    {0.38268, -0.92388, 0.92388, -0.38268, -0.38268, 0.92388, -0.92388, 0.38268},
    {0.19509, -0.55557, 0.83147, -0.98079, 0.98079, -0.83147, 0.55557, -0.19509}
};
const int size = 8;
```

FDCT.

```
void fdct_f32(int* in, int* out)
{
    int i, j, x, y;
    float s, ci, cj;
    for(i = 0; i < 8; i++)
    {
        for(j = 0; j < 8; j++)
        {
            if(i == 0)
                ci = C;
            else
                ci = 1;
            if(j == 0)
                cj = C;
            else
                cj = 1;
            s = 0.0;
            for(x = 0; x < size; x++)
                for(y = 0; y < size; y++)
                    s += Cosine[i][x]*Cosine[j][y]*in[x*size+y];
            s *= ci*cj*NORM;
            out[i*size+j] = (int)s;
        }
    }
}
```

IDCT.

```
void idct_f32(int* in, int* out)
{
    int i, j, x, y;
    float s, ax, ay;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j++)
        {
            s = 0.0;
            for(x = 0; x < size; x++)
            {
                for(y = 0; y < size; y++)
                {
```

```
                    if(x == 0)
                        ax = C;
                    else
                        ax = 1;
                    if(y == 0)
                        ay = C;
                    else
                        ay = 1;
                    s += ax*ay*in[x*size+y]*Cosine[x][i]*Cosine[y][j];
                }
            }
            s *= NORM;
            out[i*size+j] = (int)s;
        }
    }
}
```

Целочисленная реализация без MSA.

```
long _Fract lrCosine[8][8] = {
    {1.00000LR, 1.00000LR, 1.00000LR, 1.00000LR, 1.00000LR, 1.00000LR, 1.00000LR, 1.00000LR},
    {0.98079LR, 0.83147LR, 0.55557LR, 0.19509LR, -0.19509LR, -0.55557LR, -0.83147LR, -0.98079LR},
    {0.92388LR, 0.38268LR, -0.38268LR, -0.92388LR, -0.92388LR, -0.38268LR, 0.38268LR, 0.92388LR},
    {0.83147LR, -0.19509LR, -0.98079LR, -0.55557LR, 0.55557LR, 0.98079LR, 0.19509LR, -0.83147LR},
    {0.70711LR, -0.70711LR, -0.70711LR, 0.70711LR, 0.70711LR, -0.70711LR, -0.70711LR, 0.70711LR},
    {0.55557LR, -0.98079LR, 0.19509LR, 0.83147LR, -0.83147LR, -0.19509LR, 0.98079LR, -0.55557LR},
    {0.38268LR, -0.92388LR, 0.92388LR, -0.38268LR, -0.38268LR, 0.92388LR, -0.92388LR, 0.38268LR},
    {0.19509LR, -0.55557LR, 0.83147LR, -0.98079LR, 0.98079LR, -0.83147LR, 0.55557LR, -0.19509LR}
};
const int size = 8;
```

FDCT.

```
void fdct_q31(int* in, int* out)
{
    int i, j, x, y;
    long _Fract ci, cj;
    long _Accum s;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j++)
        {
            if(i == 0)
                ci = 0.70711LR;
            else
                ci = 1.0LR;
            if(j == 0)
                cj = 0.70711LR;
            else
                cj = 1.0LR;
            s = 0.0LK;
            for(x = 0; x < size; x++)
                for(y = 0; y < size; y++)
                    s += lrCosine[i][x]*lrCosine[j][y]*(long)in[x*size+y];
            s = (s*ci*cj) / 4;
            out[i*size+j] = (int)s;
        }
    }
}
```

IDCT.

```
void idct_q31(int* in, int* out)
{
    int i, j, x, y;
    long _Fract ax, ay;
    long _Accum s;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j++)
        {
            s = 0.0LK;
            for(x = 0; x < size; x++)
```

```
            {
                for(y = 0; y < size; y++)
                {
                    if(x == 0)
                        ax = 0.70711LR;
                    else
                        ax = 1.0LR;

                    if(y == 0)
                        ay = 0.70711LR;
                    else
                        ay = 1.0LR;

                    s += ax*ay*in[x*size+y]*lrCosine[x][i]*lrCosine[y][j];
                }
            }
            s = s / 4;
            out[i*size+j] = (int)s;
        }
    }
}
```

Реализация с MSA с использованием чисел с плавающей точкой.

```
#define NORM 0.25
#define C 0.70711
float Cosine[8][8] = {
    {1.00000, 1.00000, 1.00000, 1.00000, 1.00000, 1.00000, 1.00000, 1.00000},
    {0.98079, 0.83147, 0.55557, 0.19509, -0.19509, -0.55557, -0.83147, -0.98079},
    {0.92388, 0.38268, -0.38268, -0.92388, -0.92388, -0.38268, 0.38268, 0.92388},
    {0.83147, -0.19509, -0.98079, -0.55557, 0.55557, 0.98079, 0.19509, -0.83147},
    {0.70711, -0.70711, -0.70711, 0.70711, 0.70711, -0.70711, -0.70711, 0.70711},
    {0.55557, -0.98079, 0.19509, 0.83147, -0.83147, -0.19509, 0.98079, -0.55557},
    {0.38268, -0.92388, 0.92388, -0.38268, -0.38268, 0.92388, -0.92388, 0.38268},
    {0.19509, -0.55557, 0.83147, -0.98079, 0.98079, -0.83147, 0.55557, -0.19509}
};
const int size = 8;
```

FDCT.

```
void msa_fdct_f32(int* in, int* out)
{
    int i, j, x, y;
    v4f32 ci, cj;
    v4f32 s;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j+=4)
        {
            if(i == 0)
                ci = (v4f32) {0.70711f, 0.70711f, 0.70711f, 0.70711f};
            else
                ci = (v4f32) {1.0f, 1.0f, 1.0f, 1.0f};
            if(j == 0)
                cj = (v4f32) {0.70711f, 1.0f, 1.0f, 1.0f};
            else
                cj = (v4f32) {1.0f, 1.0f, 1.0f, 1.0f};
            s = (v4f32) {0.0f, 0.0f, 0.0f, 0.0f};
            for(x = 0; x < size; x++)
            {
                for(y = 0; y < size; y++)
                {
                    v4f32 vcos0, vcos1, vcos2, pixs;
                    pixs = (v4f32) {(float)in[x*size+y], (float)in[x*size+y],
                            (float)in[x*size+y], (float)in[x*size+y]};
                    vcos1 = (v4f32) { Cosine[i][x], Cosine[i][x],
                            Cosine[i][x], Cosine[i][x] };
                    vcos2 = (v4f32) { Cosine[j][y], Cosine[j+1][y],
                            Cosine[j+2][y], Cosine[j+3][y] };
                    vcos0 = vcos1*vcos2;
                    s = __builtin_msa_fmadd_w(s, pixs, vcos0);
                }
```

```
            }
            s = (s*ci*cj) / 4;
            out[i*size+j] = (int)s[0];
            out[i*size+j+1] = (int)s[1];
            out[i*size+j+2] = (int)s[2];
            out[i*size+j+3] = (int)s[3];
        }
    }
}
```

IDCT.

```
void msa_idct_f32(int* in, int* out)
{
    int i, j, x, y;
    v4f32 ax, ay, s;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j+=4)
        {
            s = (v4f32) {0.0f, 0.0f, 0.0f, 0.0f};
            for(x = 0; x < size; x++)
            {
                for(y = 0; y < size; y++)
                {
                    if(x == 0)
                        ax = (v4f32) { 0.70711f, 0.70711f, 0.70711f, 0.70711f };
                    else
                        ax = (v4f32) { 1.0f, 1.0f, 1.0f, 1.0f };
                    if(y == 0)
                        ay = (v4f32) { 0.70711f, 0.70711f, 0.70711f, 0.70711f };
                    else
                        ay = (v4f32) { 1.0f, 1.0f, 1.0f, 1.0f };
                    v4f32 vcos0, vcos1, vcos2, a;
                    vcos1 = (v4f32) { Cosine[x][i], Cosine[x][i],
                            Cosine[x][i], Cosine[x][i] };
                    vcos2 = (v4f32) { Cosine[y][j], Cosine[y][j+1],
                            Cosine[y][j+2], Cosine[y][j+3] };
                    vcos0 = vcos1*vcos2;
                    a = ax*ay*(float)in[x*size+y];
                    s = __builtin_msa_fmadd_w(s, a, vcos0);
                }
            }
            s = s / 4;
            out[i*size+j] = (int)s[0];
            out[i*size+j+1] = (int)s[1];
            out[i*size+j+2] = (int)s[2];
            out[i*size+j+3] = (int)s[3];
        }
    }
}
```

Целочисленная реализация с MSA.

Поскольку целочисленная реализация вносит некоторую погрешность, а использование небольшой разрядности чисел с высокими требованиями к точности возможно переполненме, то целесообразно рассмотреть разные варианты реализации алгоритма. Если требуется высокая точность вычислений, то целесообразно использовать разрядность 64-бита (при этом векторы MSA будут состоять из двух 64-битных элементов v2i64); иначе целесообразно использовать разрядность 32-бита, что обеспечит возможность работы с векторами, состоящих из четырех 32-битных элементов v4i32).

При целочисленной реализации для обеспечения требуемой точности использовался битовый сдвиг на определенное количество разрядов. Реалиация целочисленных операций для вещественных чисел описана в следующих статьях: https://habrahabr.ru/sandbox/44727/ и https://habrahabr.ru/post/131171/.

```
#define BITS 20 //точность
```

```
const int intCosine[8][8] =
{
    { (int)(1.00000000 * (1 << BITS)), (int)(1.00000000 * (1 << BITS)), (int)(1.00000000 * (1 << BITS)),
(int)(1.00000000 * (1 << BITS)), (int)(1.00000000 * (1 << BITS)), (int)(1.00000000 * (1 << BITS)),
(int)(1.00000000 * (1 << BITS)), (int)(1.00000000 * (1 << BITS)) },
    { (int)(0.98078528 * (1 << BITS)), (int)(0.83146961 * (1 << BITS)), (int)(0.55557023 * (1 << BITS)),
(int)(0.19509032 * (1 << BITS)), (int)(-0.19509032 * (1 << BITS)), (int)(-0.55557023 * (1 << BITS)),
(int)(-0.83146961 * (1 << BITS)), (int)(-0.98078528 * (1 << BITS)) },
    { (int)(0.92387953 * (1 << BITS)), (int)(0.38268343 * (1 << BITS)), (int)(-0.38268343 * (1 << BITS)),
(int)(-0.92387953 * (1 << BITS)), (int)(-0.92387953 * (1 << BITS)), (int)(-0.38268343 * (1 << BITS)),
(int)(0.38268343 * (1 << BITS)), (int)(0.92387953 * (1 << BITS)) },
    { (int)(0.83146961 * (1 << BITS)), (int)(-0.19509032 * (1 << BITS)), (int)(-0.98078528 * (1 << BITS)),
(int)(-0.55557023 * (1 << BITS)), (int)(0.55557023 * (1 << BITS)), (int)(0.98078528 * (1 << BITS)),
(int)(0.19509032 * (1 << BITS)), (int)(-0.83146961 * (1 << BITS)) },
    { (int)(0.70710678 * (1 << BITS)), (int)(-0.70710678 * (1 << BITS)), (int)(-0.70710678 * (1 << BITS)),
(int)(0.70710678 * (1 << BITS)), (int)(0.70710678 * (1 << BITS)), (int)(-0.70710678 * (1 << BITS)),
(int)(-0.70710678 * (1 << BITS)), (int)(0.70710678 * (1 << BITS)) },
    { (int)(0.55557023 * (1 << BITS)), (int)(-0.98078528 * (1 << BITS)), (int)(0.19509032 * (1 << BITS)),
(int)(0.83146961 * (1 << BITS)), (int)(-0.83146961 * (1 << BITS)), (int)(-0.19509032 * (1 << BITS)),
(int)(0.98078528 * (1 << BITS)), (int)(-0.55557023 * (1 << BITS)) },
    { (int)(0.38268343 * (1 << BITS)), (int)(-0.92387953 * (1 << BITS)), (int)(0.92387953 * (1 << BITS)),
(int)(-0.38268343 * (1 << BITS)), (int)(-0.38268343 * (1 << BITS)), (int)(0.92387953 * (1 << BITS)),
(int)(-0.92387953 * (1 << BITS)), (int)(0.38268343 * (1 << BITS)) },
    { (int)(0.19509032 * (1 << BITS)), (int)(-0.55557023 * (1 << BITS)), (int)(0.83146961 * (1 << BITS)),
(int)(-0.98078528 * (1 << BITS)), (int)(0.98078528 * (1 << BITS)), (int)(-0.83146961 * (1 << BITS)),
(int)(0.55557023 * (1 << BITS)), (int)(-0.19509032 * (1 << BITS)) }
};
const int divider = 1 << BITS;
const int c = (int)(0.70710678 * (1 << BITS));
```

FDCT v4i32.

```
void msa_fdct_i32(int* in, int* out)
{
    int i, j, x, y;
    v4i32 ci, cj, s;
    for(i = 0; i < 8; i++)
    {
        for(j = 0; j < 8; j+=4)
        {
            if(i == 0)
                ci = (v4i32) {c, c, c, c};
            else
                ci = (v4i32) {divider, divider, divider, divider};
            if(j == 0)
                cj = (v4i32) {c, divider, divider, divider};
            else
                cj = (v4i32) {divider, divider, divider, divider};
            s = (v4i32){0, 0, 0, 0};
            for(x = 0; x < size; x++)
                for(y = 0; y < size; y++)
                {
                    v4i32 vcos0, vcos1, vcos2, pixs;
                    vcos1 = (v4i32) { intCosine[i][x], intCosine[i][x],
                            intCosine[i][x], intCosine[i][x] };
                    vcos2 = (v4i32) { intCosine[j][y], intCosine[j+1][y],
                            intCosine[j+2][y], intCosine[j+3][y] };
                    vcos0 = vcos1*vcos2;
                    vcos0 = vcos0 >> BITS;
                    s += vcos0*in[x*size+y];
                }
            s = s >> BITS;
            s = (s*((ci*cj) >> BITS)) >> BITS;
            s = s >> 2;
            out[i*size+j] = s[0];
            out[i*size+j+1] = s[1];
            out[i*size+j+2] = s[2];
            out[i*size+j+3] = s[3];
        }
    }
}
```

IDCT v4i32.

```
void msa_idct_i32(int* in, int* out)
{
    int i, j, x, y;
    v4i32 s, ax, ay;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j+=4)
        {
            s = (v4i32) {0, 0, 0, 0};
            for(x = 0; x < size; x++)
            {
                for(y = 0; y < size; y++)
                {
                    if(x == 0)
                        ax = (v4i32) {c, c, c, c};
                    else
                        ax = (v4i32) {divider, divider, divider, divider};
                    if(y == 0)
                        ay = (v4i32) {c, c, c, c};
                    else
                        ay = (v4i32) {divider, divider, divider, divider};
                    v4i32 vcos0, vcos1, vcos2, a;
                    vcos1 = (v4i32) { intCosine[x][i], intCosine[x][i],
                        intCosine[x][i], intCosine[x][i] };
                    vcos2 = (v4i32) { intCosine[y][j], intCosine[y][j+1],
                        intCosine[y][j+2], intCosine[y][j+3] };
                    vcos0 = vcos1*vcos2 >> BITS;
                    a = ax*ay >> BITS;
                    a *= in[x*size+y];
                    s += a*vcos0 >> BITS;
                }
            }
            s = s >> (BITS + 2);
            out[i*size+j] = s[0];
            out[i*size+j+1] = s[1];
            out[i*size+j+2] = s[2];
            out[i*size+j+3] = s[3];
        }
    }
}


const long longCosine[8][8] =
{
    { (long)(1.00000000 * (1 << BITS)), (long)(1.00000000 * (1 << BITS)), (long)(1.00000000 * (1 <<
BITS)), (long)(1.00000000 * (1 << BITS)), (long)(1.00000000 * (1 << BITS)), (long)(1.00000000 * (1 <<
BITS)), (long)(1.00000000 * (1 << BITS)), (long)(1.00000000 * (1 << BITS)) },
    { (long)(0.98078528 * (1 << BITS)), (long)(0.83146961 * (1 << BITS)), (long)(0.55557023 * (1 <<
BITS)), (long)(0.19509032 * (1 << BITS)), (long)(-0.19509032 * (1 << BITS)), (long)(-0.55557023 * (1 <<
BITS)), (long)(-0.83146961 * (1 << BITS)), (long)(-0.98078528 * (1 << BITS)) },
    { (long)(0.92387953 * (1 << BITS)), (long)(0.38268343 * (1 << BITS)), (long)(-0.38268343 * (1 <<
BITS)), (long)(-0.92387953 * (1 << BITS)), (long)(-0.92387953 * (1 << BITS)), (long)(-0.38268343 * (1 <<
BITS)), (long)(0.38268343 * (1 << BITS)), (long)(0.92387953 * (1 << BITS)) },
    { (long)(0.83146961 * (1 << BITS)), (long)(-0.19509032 * (1 << BITS)), (long)(-0.98078528 * (1 <<
BITS)), (long)(-0.55557023 * (1 << BITS)), (long)(0.55557023 * (1 << BITS)), (long)(0.98078528 * (1 <<
BITS)), (long)(0.19509032 * (1 << BITS)), (long)(-0.83146961 * (1 << BITS)) },
    { (long)(0.70710678 * (1 << BITS)), (long)(-0.70710678 * (1 << BITS)), (long)(-0.70710678 * (1 <<
BITS)), (long)(0.70710678 * (1 << BITS)), (long)(0.70710678 * (1 << BITS)), (long)(-0.70710678 * (1 <<
BITS)), (long)(-0.70710678 * (1 << BITS)), (long)(0.70710678 * (1 << BITS)) },
    { (long)(0.55557023 * (1 << BITS)), (long)(-0.98078528 * (1 << BITS)), (long)(0.19509032 * (1 <<
BITS)), (long)(0.83146961 * (1 << BITS)), (long)(-0.83146961 * (1 << BITS)), (long)(-0.19509032 * (1 <<
BITS)), (long)(0.98078528 * (1 << BITS)), (long)(-0.55557023 * (1 << BITS)) },
    { (long)(0.38268343 * (1 << BITS)), (long)(-0.92387953 * (1 << BITS)), (long)(0.92387953 * (1 <<
BITS)), (long)(-0.38268343 * (1 << BITS)), (long)(-0.38268343 * (1 << BITS)), (long)(0.92387953 * (1 <<
BITS)), (long)(-0.92387953 * (1 << BITS)), (long)(0.38268343 * (1 << BITS)) },
    { (long)(0.19509032 * (1 << BITS)), (long)(-0.55557023 * (1 << BITS)), (long)(0.83146961 * (1 <<
BITS)), (long)(-0.98078528 * (1 << BITS)), (long)(0.98078528 * (1 << BITS)), (long)(-0.83146961 * (1 <<
BITS)), (long)(0.55557023 * (1 << BITS)), (long)(-0.19509032 * (1 << BITS)) }
};
```

```
const long ldivider = (long)(1 << BITS);
const long lc = (long)(0.70710678 * (1 << BITS));
```

FDCT v2i64.

```
void msa_fdct_i64(int* in, int* out)
{
    int i, j, x, y;
    v2i64 ci, cj, s;

    for(i = 0; i < 8; i++)
    {
        for(j = 0; j < 8; j+=2)
        {
            if(i == 0)
                ci = (v2i64) {lc, lc};
            else
                ci = (v2i64) {ldivider, ldivider};

            if(j == 0)
                cj = (v2i64) {lc, ldivider};
            else
                cj = (v2i64) {ldivider, ldivider};

            s = (v2i64){0, 0};

            for(x = 0; x < size; x++)
                for(y = 0; y < size; y++)
                {
                    v2i64 vcos0, vcos1, vcos2, pixs;
                    vcos1 = (v2i64) { longCosine[i][x], longCosine[i][x] };
                    vcos2 = (v2i64) { longCosine[j][y], longCosine[j+1][y] };

                    vcos0 = vcos1*vcos2;
                    vcos0 = vcos0 >> BITS;
                    s += vcos0*(long)in[x*size+y];
                }
            s = s >> BITS;
            s = (s*((ci*cj) >> BITS)) >> BITS;
            s = s >> 2;
            out[i*size+j] = s[0];
            out[i*size+j+1] = s[1];
        }
    }
}
```

IDCT v2i64.

```
void msa_idct_i64(int* in, int* out)
{
    int i, j, x, y;
    v2i64 s, ax, ay;
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size; j+=2)
        {
            s = (v2i64) {0, 0};
            for(x = 0; x < size; x++)
            {
                for(y = 0; y < size; y++)
                {
                    if(x == 0)
                        ax = (v2i64) {lc, lc};
                    else
                        ax = (v2i64) {ldivider, ldivider};
                    if(y == 0)
                        ay = (v2i64) {lc, lc};
                    else
                        ay = (v2i64) {ldivider, ldivider};
                    v2i64 vcos0, vcos1, vcos2, a;
                    vcos1 = (v2i64) { longCosine[x][i], longCosine[x][i]};
                    vcos2 = (v2i64) { longCosine[y][j], longCosine[y][j+1]};
```

```
                    vcos0 = vcos1*vcos2 >> BITS;
                    a = ax*ay >> BITS;
                    a *= in[x*size+y];
                    s += a*vcos0 >> BITS;
                }
            }
            s = s >> BITS;
            s = s >> 2;
            out[i*size+j] = s[0];
            out[i*size+j+1] = s[1];
        }
    }
}
```

Сравнение точности различных реализаций.

Поскольку использование целочисленных операция для работы с вещественными числами внесет некоторую погрешность в выходные данные, то необходимо оценить рациональность использования целочисленной арифметики, исследовав точность работы алгоритма. Для оценки точности работы алгоритма используется среднеквадратическое отклонение от эталона. В качестве эталона используется обычная реализация с плавающей запятой.

FDCT.

В данной таблице приведено СКО от эталона для каждого алгоритма по прямому дискретному косинусному преобразование.

СКО при целочисленной реализации с фиксированной точкой без MSA fdct_q31 (точность 2^{31}).	0
СКО при реализации с плавающей точкой с MSA msa_fdct_f32.	0
СКО при целочисленной реализации с MSA msa_fdct_i32 (точность 2^{10}).	1.7185
СКО при целочисленной реализации с MSA msa_fdct_i64 (точность 2^{20}).	0.85696

IDCT.

В данной таблице приведено СКО от эталона для каждого алгоритма по обратному дискретному косинусному преобразование.

СКО при целочисленной реализации с фиксированной точкой без MSA idct_q31 (точность 2^{31}).	0
СКО при реализации с плавающей точкой с MSA msa_idct_f32.	0
СКО при целочисленной реализации с MSA msa_idct_i32 (точность 2^{10}).	0.625
СКО при целочисленной реализации с MSA msa_idct_i64 (точность 2^{20}).	0

Использование инструкций MSA.

В данном разделе будет отображено кол-во и название обычных инструкций и инструкций MSA, которые использовались в программе.

Перед выполнением программы происходит ее развертывание (настройка адресации, построение PSP и т.д.), поэтому число вызовов обычных операций не точное. Для определения примерного количества операций для развертывания было подсчитано их количество при теле main функции return 0. Из этого получилось, что для развертывания программы необходимо 12867 операции.

Ниже представлены таблицы с подсчетом команд в различных вариантах реализации алгоритма. Операции с префиксом OPC являются обычными, все остальные – это MSA команды. Перечень MSA команд можно посмотреть в прилагаемом PDF файле.

Реализация с плавающей точной с MSA msa_fdct_f32.

Команда	Количество вызовов
OPC_LOGIC	4864
OPC_LOGIC_IMM	550
OPC_ARITH_IMM	12704
OPC_ARITH	25203
OPC_LD	49585
OPC_ST	2721
OPC_COND_MOVE	40
OPC_SLT	14
OPC_SHIFT_IMM	21417
OPC_SLT_IMM	111
OPC_ST_COND	8
OPC_SHIFT	9
OPC_COPL_LDST	8264
OPC_FLT_LDST	8264
LD.df	5344
ST.df	5184
INSERT.df	18464
FMUL.df	1056
FMADD.df	1024
FDIV.df	16

Количество обычных инструкций: 120890
Количество инструкций MSA: 31088
Процент инструкций MSA: 20%

Целочисленная реализация с MSA msa_fdct_i32 (точность 2^{10}).

OPC_LOGIC	11008
OPC_LOGIC_IMM	716
OPC_ARITH_IMM	9584
OPC_ARITH	19059
OPC_LD	45483
OPC_ST	2721
OPC_COND_MOVE	40
OPC_SLT	14
OPC_SHIFT_IMM	18345
OPC_SLT_IMM	111
OPC_ST_COND	8
OPC_SHIFT	9
OPC_COPL_LDST	8
OPC_FLT_LDST	8
LDI.df	2096
INSERT.df	8320
ST.df	5216

LD.df	5328
MULV.df	2080
SRAI.df	1088
FILL.df	1024
MOVE.V	2048
ADDV.df	1024

Количество обычных инструкций: 94247

Количество инструкций MSA: 28224

Процент инструкций MSA: 23%

Целочисленная реализация с MSA msa_fdct_i64 (точность 2^{20}).

OPC_LOGIC	7936
OPC_LOGIC_IMM	746
OPC_ARITH_IMM	12048
OPC_ARITH	21107
OPC_LD	53131
OPC_ST	4033
OPC_COND_MOVE	40
OPC_SLT	14
OPC_SHIFT_IMM	19433
OPC_SLT_IMM	111
OPC_ST_COND	8
OPC_SHIFT	9
OPC_COPL_LDST	8
OPC_FLT_LDST	8
LDI.df	4192
INSERT.df	8320
ST.df	10432
LD.df	10656
MULV.df	4160
SRAI.df	2176
FILL.df	2048
MOVE.V	4096
ADDV.df	2048

Количество обычных инструкций: 105765

Количество инструкций MSA: 48128

Процент инструкций MSA: 31%

Реализация с плавающей точной с MSA msa_idct_f32.

OPC_LOGIC	1792
OPC_LOGIC_IMM	550
OPC_ARITH_IMM	11648
OPC_ARITH	19059
OPC_LD	43099
OPC_ST	2737
OPC_COND_MOVE	40
OPC_SLT	14
OPC_SHIFT_IMM	18345
OPC_SLT_IMM	111
OPC_ST_COND	8
OPC_SHIFT	9
OPC_COPL_LDST	8360
OPC_FLT_LDST	8360
LD.df	9376
ST.df	7184
INSERT.df	12288
FMUL.df	3072
SPLATI.df	1024
FMADD.df	1024

FDIV.df	16

Количество обычных инструкций: 101265

Количество инструкций MSA: 33984

Процент инструкций MSA: 25%

Целочисленная реализация с MSA msa_idct_i32 (точность 2^{10}).

OPC_LOGIC	7936
OPC_LOGIC_IMM	11558
OPC_ARITH_IMM	9584
OPC_ARITH	19059
OPC_LD	47499
OPC_ST	2721
OPC_COND_MOVE	40
OPC_SLT	14
OPC_SHIFT_IMM	18345
OPC_SLT_IMM	111
OPC_ST_COND	8
OPC_SHIFT	9
OPC_COPL_LDST	8
OPC_FLT_LDST	8
LDI.df	4112
ST.df	8224
LD.df	8336
INSERT.df	16384
MULV.df	4096
SRAI.df	3088
FILL.df	1024
MOVE.V	1024
ADDV.df	1024

Количество обычных инструкций: 104033

Количество инструкций MSA: 47312

Процент инструкций MSA: 31%

Целочисленная реализация с MSA msa_idct_i64 (точность 2^{20}).

OPC_LOGIC	14080
OPC_LOGIC_IMM	13350
OPC_ARITH_IMM	12048
OPC_ARITH	21107
OPC_LD	57163
OPC_ST	4033
OPC_COND_MOVE	40
OPC_SLT	14
OPC_SHIFT_IMM	19433
OPC_SLT_IMM	111
OPC_ST_COND	8
OPC_SHIFT	9
OPC_COPL_LDST	8
OPC_FLT_LDST	8
LDI.df	10272
ST.df	16480
LD.df	16704
INSERT.df	20480
MULV.df	8192
SRAI.df	6208
MOVE.V	2048
ADDV.df	2048

Количество обычных инструкций: 128545

Количество инструкций MSA: 82432

Процент инструкций MSA: 39%

Рекомендации по реализации на MSA

- Минимизировать использование обычных инструкций
- Использовать векторные операции для загрузки/извлечения векторов из памяти.
- Использовать операцию умножения с накоплением MADD
- Для более быстрого копирования косинусов в вектор MSA создать 2 отдельных массива, данные в которых структурированы таким образом, чтобы при инициализации вектора косинусов брались соседние элементы. Для прямого и обратного преобразования нужны отдельные массивы (всего нужно 2)
- Коэффициенты C_i, C_j, A_x, A_y задать как константы.

Рекомендации по введению новых инструкций

- Быстрое копирование MSA вектора в обычный массив.
- Операция умножение со сдвигом для векторов.